Chatty: simple design, interesting problems

Xavier León

Introduction

Your task will be to implement a distributed system that will allow you to chat among buddies. The purpose of this seminar is that you think about the main problems in distributed systems as well as learn a little bit of Erlang. We are going to implement two different versions of our chat system: i) a system composed by a single chat server to chat with your buddies; ii) a decentralized system with several chat servers which allows to clients connected to different servers chat with each other.

This document provides almost every piece of code. However, you may need to fill in the gaps (...) to ensure you understand what is the expected behaviour of the system. We encourage you to write down the code by yourself (NOT copy&paste) to get used to the Erlang syntax and structure sooner! In the following seminars, the code provided won't be so detailed.

1 Chatting with buddies

The initial version of the chat will consist of two different types of processes: *clients* (friend to chat with) and a *server*. Clients will connect –and, of course, disconnect– to the server and the server is going to be responsible to maintain the list of clients attached and relay messages sent by a client to the rest of clients. As this design is quite simple from the distributed systems point of view, we are going make things more robust later on.

1.1 The server

We need to implement a server that keeps track of connected users and relay messages sent by one user to the rest of users. Thus, the message interface the server is going to handle is as follows:

- `{client_join_req Name, From}`: a join request from a client containing its username (Name) and a reference or process identifier (From) that the server should use to contact it. We need to update the list of connected clients and send to the connected users this new event (join).
- `{client_leave_req Name, From}`: a leave request from a client containing its username (Name) and a reference to its handler process (From). We need to update (remove) the reference from the list of connected clients and send to the connected users this new event (leave).

- `{send, Name, Text}`: a request to send a message (Text) to the connected users and the username (Name).

The following code is an example of the implementation of the above message interface (remember to fill the gaps). Open up a new file **server.erl** and declare the module **server**:

```
-module(server).
%% Exported Functions
-export([start/0, process_requests/1]).

%% API Functions
start() ->
    ServerPid = spawn(server, process_requests, [[]]),
    register(server_process, ServerPid).

process_requests(Clients) ->
    receive
        {client_join_req, Name, From} ->
            NewClients = [...|Clients],
            broadcast(NewClients, {join, Name}),
            process_requests(...);
        {client_leave_req, Name, From} ->
            NewClients = lists:delete(..., Clients),
            broadcast(Clients, {leave, Name}),
            process_requests(...);
        {send, Name, Text} ->
            broadcast(Clients, {message, Name, Text}),
            process_requests(...)
    end.

%% Local Functions
broadcast(PeerList, Message) ->
    Fun = fun(Peer) -> Peer ! Message end,
    lists:map(Fun, PeerList).
```

1.2 The client

So far, we have seen how the server is implemented. Now, we will specify how the client works. The client process will have two different tasks to perform. We will have a background task responsible for handling replies from the server and the main task will be responsible to read a message from the standard input to be sent to the rest of your buddies. The background task (the one handling server replies) should handle the following message interface:

- `{join, Name}`: the server informs that a new client is connected. We should write through the standard output this information.
- `{leave, Name}`: the server informs that a connected client is about to disconnect. We should write down this information through the standard output.

- `{message, Name, Text}`: this is a message sent back by the server from one of the connected users (Name). We should print this message (Text) so we can actually read it.

Notice that, in this case, we do not need the reference to the client sending a message (From) as a buddy does not contact directly with other buddies but with the server.

The main process should block waiting for the user to write down some text to send to the chatting room. If a user wants to leave, we need to write a *command* (exit) to actually leave the chat. Once the user writes this keyword, the main process will send a request to the server to leave the room.

The following code is an example of the implementation of the above message interface (remember to fill the gaps). Open a new file **client.erl** and declare the module **client**.

```
-module(client).

%% Exported Functions
-export([start/2, init_client/2]).

%% API Functions
start(ServerPid, MyName) ->
    ClientPid = spawn(client, init_client, [ServerPid, MyName]),
    process_commands(ServerPid, MyName, ClientPid).

init_client(ServerPid, MyName) ->
    ServerPid ! {client_join_req, MyName, self()},
    process_requests().

%% Local Functions
%% This is the background task logic
process_requests() ->
    receive
        {join, Name} ->
            io:format("[JOIN] ~s joined the chat~n", [Name]),
            ...;
        {leave, Name} ->
            io:format("[LEAVE] ~s leaves the chat~n", [Name]),
            process_requests();
        {message, Name, Text} ->
            io:format("[~s] ~s", [Name, Text]),
            ...
    end.
```

```
%% This is the main task logic
process_commands(ServerPid, MyName, ClientPid) ->
    %% Read from standard input and send to server
    Text = io:get_line("-> "),
    if
        Text  == "exit\n" ->
            ServerPid ! {client_leave_req, MyName, ClientPid};
        true ->
            ServerPid ! {send, MyName, Text},
            process_commands(ServerPid, MyName, ClientPid)
    end.
```

1.3 Testing

Debugging Erlang code may be quite confusing sometimes because of the error messages. Once you get used to them, you will quickly understand what is wrong with your code. In the meanwhile, it is useful to put debugging information to know where your code is failing and the state of your variables.

If you are not using an Erlang IDE which automatically compiles your code, you will need to do it by hand –every time you modify the file– before calling your implemented procedures. A simple test can be done in a single computer. You will execute a server and a client in different terminals. To start a server type (remember to change the local IP by your public IP when comunicating with remote nodes):

```
user@host:~/Chatty$ erl -name server_node@127.0.0.1 -setcookie secret

(server_node@127.0.0.1)1> c(server). %% Compile server module
(server_node@127.0.0.1)2> server:start().
true
```

Open up a new terminal and write the following:

```
user@host:~/Chatty$ erl -name client_node@127.0.0.1 -setcookie secret

(client_node@127.0.0.1)1> c(client). %% Compile client module
(client_node@127.0.0.1)2> client:start({server_process, 'server_node@127.0.0.1'}, "John").
[JOIN] John joined the chat
```

And if you type a message, you should see something like...

```
-> hi!
[John] hi!
```

Try opening 2 or 3 clients and test that all of them receive all the messages. You can also try to communicate among different physical machines (remember to change the local IP line by the public IP of the server you are connecting to). Read the Erlang primer to refresh how to contact remote nodes.

Open Questions. What are the advantages and disadvantages of this implementation? What happens if the server fail? Are the messages guaranteed to be delivered in order? Does it matter the order in which users join and leave the chat? Does this solution scale when the number of users increase?

2 Making it robust

The previous design has its disadvantages. Mainly, it is not robust to failures –i.e. if the server fails the whole system will become useless. A solution to this problem is to have more servers (replication). Thus, if a server fails, we will have other servers still running and continue sending messages. As usually, solving a problem is an open door for other interesting problems.

We will have a set of replicated servers to which users may connect indistinctly. This way, if a server fails, only those users connected to the failing server will lose connectivity but the rest will remain connected. Of course we could implement a solution in which clients automatically reconnect to another server when they detect a failure but we are going to keep things simple.

We will only need to change the server implementation to introduce this new functionality. This new implementation needs to know the list of replicated servers. Besides, we need to handle the membership of the set of servers. Users will connect to one of the servers and send messages to it. This server will forward the message to the set of servers which in turn will relay the message to its clients.

The new messages that the server needs to handle are as follows:

- **{**`server_join_req, From`**}**: a new server is added to the set of replicated servers. We need to update the list of servers and inform the rest of servers of this new node.
- **{**`update_servers, NewServers`**}**: this message is received when a new server is added to the set.
- `RelayMessage`: if the message received does not match any of the previous clauses, the server relay the message to all of its clients (it may be either a message of type `join`, `leave` or `message`).

Be aware that now, the messages previously implemented (`client_join_req`, `client_leave_req` and `send`) are not forwarded directly to the clients but to every member of the set of servers. They in turn will relay those messages to their connected clients.

The following code is an example of the implementation of the above message interface (remember to fill the gaps). Open a new file **server2.erl** and declare the module **server2**.

```
-module(server2).

%% Exported Functions
-export([start/0, start/1, init_server/0, init_server/1]).

%% API Functions
start() ->
    ServerPid = spawn(server2, init_server, []),
    register(server_process, ServerPid).

start(BootServer) ->
    ServerPid = spawn(server2, init_server, [BootServer]),
    register(server_process, ServerPid).
```

```
init_server() ->
    process_requests([], [self()]).

init_server(BootServer) ->
    BootServer ! {server_join_req, self()},
    process_requests([], []).

process_requests(Clients, Servers) ->
    receive
        %% Messages between client and server
        {client_join_req, Name, From} ->
            NewClients = [From|Clients],
            broadcast(..., {join, Name}),
            process_requests(NewClients, Servers);
        {client_leave_req, Name, From} ->
            NewClients = lists:delete(From, Clients),
            broadcast(..., {leave, Name}),
            process_requests(NewClients, Servers);
        {send, Name, Text} ->
            broadcast(Servers, {message, Name, Text}),
            process_requests(Clients, Servers);

        %% Message between servers
        {server_join_req, From} ->
            NewServers = [From| Servers],
            broadcast(..., {update_servers, NewServers}),
            process_requests(Clients, NewServers);
        {update_servers, NewServers} ->
            io:format("[SERVER UPDATE] ~w~n", [NewServers]),
            process_requests(Clients, ...);
        RelayMessage -> %% Whatever other message is relayed to its clients
            broadcast(Clients, RelayMessage),
            process_requests(Clients, Servers)
    end.

%% Local Functions

broadcast(PeerList, Message) ->
    Fun = fun(Peer) -> Peer ! Message end,
    lists:map(Fun, PeerList).
```

The architecture of our decentralized system is shown in figure 1.You need to first start a server with the function `server2:start()` –which creates a server with an empty list of server references– and, then start different servers with the function `server2:start({server,'server_node@IP'})`–which will connect this server to the rest of servers. Remember to change the remote hostname by the IP of the server you are connecting to.

Once you have a set of servers up and running, try connecting some clients to each of the server instances and begin to chat. Does it work? You can also

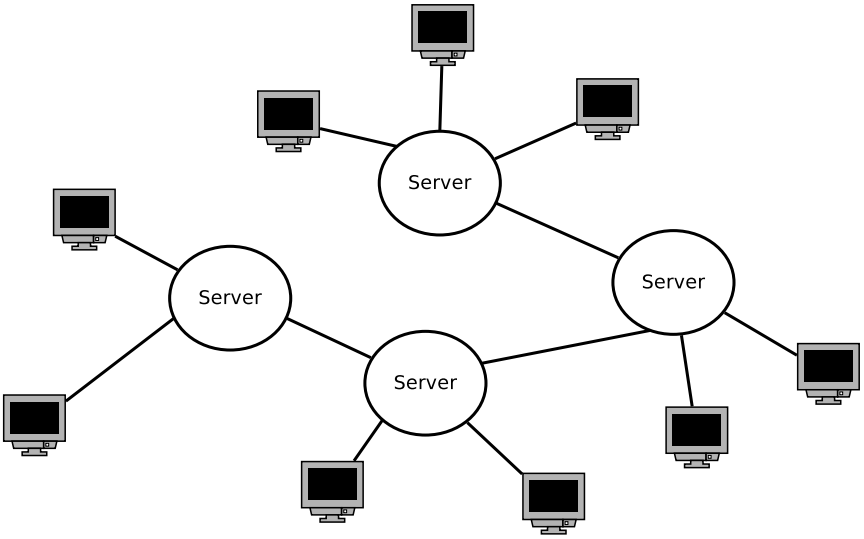

Figure 1: Architecture of decentralized system

try to crash some of the servers and see what happens.

Open Questions. What are the advantages and disadvantages of this implementation? Is it better or worse than the centralized implementation? What happens if a server fails? Are the messages guaranteed to be delivered in order? Does it matter the order and time at which users join and leave the chat? And for the servers, does it matter when a server join and leave? What happens if there are concurrent requests to join the system? Does this solution scale?

3 Report

Write up a short report (1-2 pages) with the difficulties you found during this seminar. You should discuss the pros and cons with each solution by answering or discussing the questions presented at the end of Sections 1 and 2. You should deliver the report before the next lab session (1 week).